Multi-Core Code Generation From Interface Based Hierarchy

Jonathan Piat, Shuvra S. Bhattacharyya, Maxime Pelcat, Mickaël Raulet

Multi-core code generation from Interface based hierarchy

Jonathan Piat[1], Shuvra S. Bhattacharyya[2], Maxime Pelcat[1], and Mickael Raulet[1]
[1]IETR/INSA, UMR CNRS 6164
Image and Remote Sensing laboratory
F-35043 Rennes, France
email: {firstname.lastname@insa-rennes.fr}
[2]Department of Electrical and Computer Engineering,
University of Maryland
College Park, MD, 20742, USA
email: {ssb@umd.edu}

Abstract

Dataflow has proved to be an attractive computational model for programming digital signal processing (DSP) applications. A restricted version of dataflow, termed synchronous dataflow (SDF), offers strong compile-time predictability properties, but has limited expressive power. A new type of hierarchy semantics that we propose for the SDF model allows more expressivity in SDF while maintaining its predictability. This new hierarchy semantic is based on interfaces that fix the number of tokens consumed/produced by a hierarchical vertex in a manner that is independent or separate from the specified internal dataflow structure of the encapsulated subsystem. This interface-based hierarchy gives the application designer more flexibility to apply iterative design approaches, and to make optimizing choices at the design level. This type of hierarchy is also closer to the host language semantics (i.e., the the semantics of the languages, such as C, Java and Verilog/VHDL, in which the internal functionality of primitive SDF blocks is typically written) because hierarchy levels can be interpreted as code closures (i.e., semantic boundaries), and allow one to design iterative patterns. This paper presents our proposed approach to hierarchical SDF system design, and demonstrates how we can take advantage of the proposed hierarchy semantics to generate efficient static C code targeting embedded applications.

1 Introduction

Since applications such as video coding/decoding or digital communications with advanced features are becoming more complex, the need for computational power is rapidly increasing. In order to satisfy software requirements, the use of parallel architecture is a common answer. To reduce the software development effort for such architectures, it is necessary to provide the programmer with efficient tools capable of automatically solving communications and software partitioning/scheduling concerns. Most tools such as PeaCE [14], SynDEx [3] or PREESM [9] use as an entry point a model of the application associated to a model of the architecture. Data flow model is indeed a natural representation for data-oriented applications since it represents data dependencies between the operations allowing to extract parallelism. In this model the application is described as a graph in which nodes represent computations and edges carry the stream of data-tokens between operations. The Synchronous Data Flow (SDF) model allows to specify the number of tokens produced/consumed on each outgoing/incoming edges for one firing of a node. Edges can also carry initialization tokens, called delay. That information allows to perform analysis on the graph to determine whether or not the graph is schedule-able, and if so to determine an execution order of the nodes and application's memory requirements.

In order to extend the expressivity of the SDF model, we propose a new hierarchy type more detailed in [8] allowing the designer to describe sub-graphs in a top down approach, thus adding relevant information for later optimizations. The purpose of this paper is to describe the methods allowing to generate static C code for multi-processor architecture from a given hierarchical SDF description. This C code can then be compiled for the operators of the architecture. Code generation is a key point of rapid prototyping tools as it allows the designer to go from its representation to an efficient implementation. Application such as RVC (Reconfigurable Video Coding) needs rapid proto-

typing tools allowing to re-use portion of already developed algorithm in any video coding/decoding profile. In that case static code generation leads to a predictable implementation that can easily target embedded system and takes advantage of the multi (heterogeneous) cores architectures.

Section 2 explains the data flow semantics and particularly synchronous data flow graphs, section 3 presents the existing hierarchy types and 4 introduces the proposed hierarchy and its code generation. Section 5 uses the described hierarchy type to design an example and provides some results. Finally, section 6 highlights the future work and concludes this paper.

2 Synchronous Data Flow Graph

The Synchronous Data Flow (SDF) graph [5] is used to simplify the application specification, by allowing the representation of the application behavior at a coarse grain. This data flow model represents operations of the application and specifies data dependencies between the operations.

A Synchronous Data Flow graph is a finite directed, weighted graph $G =< V, E, d, p, c >$ where :

- V is the set of nodes; each node represents a computation that operates on one or more input data streams and outputs one or more output data streams.
- $E \subseteq V \times V$ is the edge set, representing channels which carry data streams.
- $d : E \rightarrow N \cup \{0\}$ $(N = 1, 2, \ldots)$ is a function with $d(e)$ the number of initial tokens on an edge e.
- $p : E \rightarrow N$ is a function with $p(e)$ representing the number of data tokens produced at e's source to be carried by e.
- $c : E \rightarrow N$ is a function with $c(e)$ representing the number of data tokens consumed from e by e's sink node.

The topology matrix is the matrix of size $|E| \times |V|$, in which each row corresponds to an edge e in the graph and each column corresponds to a node v. Each coefficient (i, j) of the matrix is positive and equal to N if N tokens are produced by the j^{th} node on the i^{th} edge or negative and equal to N if N tokens are consumed by the j^{th} node on the i^{th} edge. It was proved in [5] that a static schedule for graph G can be computed only if its topology matrix's rank is one less than the number of nodes in G. This necessary condition means that there is a Basic Repetition Vector (BRV) q of size $|V|$ in which each coefficient is the repetition factor for the j^{th} vertex of the graph. SDF graph representation allows use of hierarchy, meaning that for $v = G$, a vertex may be described as a graph. A vertex with no hierarchy is called an actor.

2.1 SDF to DAG translation

One common way to schedule SDF graphs onto multiple processors is to first convert the SDF graph into a precedence graph such that each vertex in the precedence graph corresponds to a single execution of an actor from the SDF graph. Thus each SDF graph actor A is "expanded into" q_A separate precedence graph vertices, where q_A is the component of the BRV that corresponds to A. In general, the SDF graph exposes some of the functional parallelism in the algorithm; the precedence graph may reveal more functional parallelism, and in addition, it exposes the available data-parallelism. A valid precedence graph contains no cycle and is called DAG (Directed Acyclic Graph). Unfortunately, the expansion due to the repetition count of each SDF node can lead to an exponential growth of nodes in the DAG. Thus, precedence-graph-based multiprocessor scheduling techniques, such as those developed in [11, 12], in general have complexity that is not polynomially bounded in the size of the input SDF graph, and can result in prohibitively long scheduling times for certain kinds of graphs (e.g., see [10]).

3 Hierarchy types in SDF graph

Hierarchy can be extracted from a graph in order to optimize application for the scheduling, but can also be used by user to describe an application at different grain level. A first type of hierarchy has been described in [10], as a mean of representing cluster of actor in a SDF graph. In [10] clustering is used as a pre-pass for the scheduling described in [4] that reduces the number of vertices in the DAG, minimizes synchronization overhead for multithreaded implementation and maximizes the throughput by grouping buffers [4]. Another type of hierarchy as also been introduced in the Parameter-based SDF model. The PSDF hierarchy is described in [1] where the authors introduce a new SDF model called *Parameterized SDF*. This model aims at increasing SDF expressivity while maintaining its compile time predictability properties. In this model a subsystem (sub-graph) behavior can be controlled by a set of parameters that can be configured dynamically. These parameters can either configure sub-system interface behavior by modifying production/consumption rate on interfaces, or configure behavior by passing parameters (values) to the sub-system actors.

From a programmer view it is important to be able to design independent parts of an application (function) considering fixed interfaces and then instantiate them into the application. This allows to have a local analysis and to reuse

code into different application with minimal modifications. To do so it is necessary to define an additional kind of hierarchy which is designer/programmer oriented. This kind of hierarchy allows to keep the statical properties of the SDF while extending its expressivity by specifying the interfaces behavior.

4 Interface-based SDF Hierarchy

While designing an application, user might want to use hierarchy in a way to design independent graphs that can be instantiated in any design. From a programmer view it behaves as closures since it defines limits for a portion of an application. This kind of hierarchy must ensure that while a graph is instantiated, its behavior might not be modified by its parent graph, and that its behavior might not introduce deadlock in its parent graph. The rules defined in the composition rules ensure the graph to be deadlock free when verified, but are used to analyze a graph with no hierarchy. In order to allow the user to hierarchically design a graph, this hierarchy semantic must ensure that the composed graph will have no deadlock if every level of hierarchy is independently deadlock free. To ensure this rule we must integrate special nodes in the model that restrict the hierarchy semantic. In the following a hierarchical vertex will refer to a vertex which embeds a hierarchy level, and a sub-graph will refer to the graph representing this hierarchy level.

4.1 Special nodes

Source node: A Source node is a bounded source of tokens which represents the tokens available for an iteration of the sub-graph. This node behaves as an interface to the outside world. A source port is defined by the four following rules:

A-1 Source production homogeneity: A source node $Source$ produces the same amount of tokens on all its outgoing connections $p(e) = n \quad \forall e \in \{Source(e) = Source\}$.

A-2 Interface Scope: The source node remains write-locked during an iteration of the sub-graph. This means that the interface cannot be filled by the outside world during the sub-graph execution.

A-3 Interface boundedness: A source node cannot be repeated, thus any node consuming more tokens than made available by the node will consume the same tokens multiple times (ring buffer). $c(e)\%p(e) = 0 \quad \forall e \in \{source(e) = source\}$.

A-4 SDF consistency: All the tokens made available by a source node must be consumed during an iteration of the sub-graph.

Sink node: A sink node is a bounded sink of tokens that represent the tokens to be produced by an iteration of the graph. This node behaves as an interface to the outside world. A sink node is defined by the four following rules:

B-1 Sink producer uniqueness: A sink node $Sink$ only has one incoming connection.

B-2 Interface Scope: The sink node remains read-locked during an iteration of the sub-graph. This means that the interface cannot be read by the outside world during the sub-graph execution.

B-3 Interface boundedness: A sink node cannot be repeated, thus any node producing more tokens than needed by the node will write the same tokens multiple times (ring buffer). $p(e)\%c(e) = 0 \quad \forall e \in \{target(e) = Sink\}$.

B-4 SDF consistency: All the token consumed by a sink node must be produced during an iteration of the sub-graph.

4.2 Hierarchy deadlock-freeness

Considering a consistent connected SDF graph $G = \{g, z\}$, $g = \{Source, x, y, Sink\}$ with $Source$ being a source node and $Sink$ being a sink node, and z being an actor. In the following we show how the hierarchy rules described above ensure the hierarchical vertex g to not introduce deadlocks in the graph G:

- if it exists a simple path going from x to y containing more than one arc, this path cannot introduce cycle since this path contains at least one interface, meaning that the cycle gets broken. User must take this into account to add delay to the top graph.

- Rules **A2-B2** ensure that all the data needed for an iteration of the sub-graph are available as soon as its execution starts, and that no external vertex can consume on the sink interface while the sub-graph is being executed. As a consequence no external vertex strongly connected with the hierarchical vertex can be executed concurrently. The interface ensures the sub-graph content to be independent to the outside world, as there is no edge $\alpha \in \left\{ \alpha' \| \begin{pmatrix} (src(\alpha') = x) \\ and \\ (snk(\alpha') \in C) \\ and \\ (snk(\alpha') \notin \{x, y\}) \end{pmatrix} \right\}$ considering that $snk(\alpha') \notin \{x, y\}$) cannot happen.

- The designing approach of the hierarchy cannot lead to an hidden delay since even if a delay is in the sub-graph, an iteration of the sub-graph cannot start if its input interfaces are not full.

Those rules also guarantee that the edges of the sub-graph have a local scope, since the interfaces make the inner graph independent from the outside world. This means that when an edge in the sub-graph creates a cycle (and contains a delay), if the sub-graph needs to be repeated this iterating edge will not link multiple instances of the sub-graph.

The given rules are sufficient to ensure a sub-graph to not create deadlocks when instantiated in a larger graph.

4.3 Hierarchy scheduling

As explained in [6] interfaces to the outside world must not be taken into account to compute the schedule-ability of the graph. As in our hierarchy interpretation, interfaces have a meaning for the sub-graph, they must be taken into account to compute the schedule-ability, since we must ensure that all the tokens on the interfaces will be consumed/produced in an iteration of the sub-graph (see rules **A4-B4**).

Due to the interface nature, every connection coming/-going from/to an interface must be considered like a connection to an independent interface. Adding an edge e to graph G increases the rank of its topology matrix Γ if the row added to Γ is linearly independent from the other row. Adding an interface to a graph G composed of N vertices, and one edge e connecting this interface to G adds a linearly independent row to the topology matrix. This increases the rank of the topology matrix of one, but adding the interface's vertex will yield in a $N+1$ graph : $rank(\Gamma(G_N)) = N-1 \Rightarrow rank(\Gamma(G_{N+1})) = rank(\Gamma(G_N)) + 1 = (N+1)-1$. The rank of the topology matrix remains equal to the number of vertices less one meaning that this graph remains schedule-able. Since adding an edge between a connected interface and any vertex of the graph, results in (in meaning) adding an edge between a newly created interface and the graph, it does not affect the schedule-ability considering the proof above. This means that a sub-graph can be considered schedule-able if its actor graph (excluding interfaces) is schedule-able.

Before scheduling a hierarchical graph, we must verify that every level of hierarchy is deadlock free. Applying the balance equation to every level is sufficient to prove the deadlock freeness of a level.

4.4 Hierarchy behavior

Interfaces behavior can vary due to the graph schedule. This behavior flexibility can ease the development process, but needs to be understood to avoid meaningless applications.

- Source behavior

As said in the source interface rules, a source interface can have multiple outgoing (independent) connection and reading more tokens than made available results in reading modulo the number of tokens available (circular buffer). This means that the interface can behave like a broadcast. In Figure 1, vertices A and B have to execute respectively 4 and 6 times to consume all the data made available by the port. In this example, the $Source$ interface will broadcast twice to vertex A and three times to vertex B. This means that the designer must keep in mind that the interfaces can have effect on the inner graph schedule.

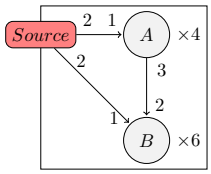


Figure 1. Source example and its execution pattern

- Sink behavior

As said in the sink interface rules, a source interface can only have one incoming connection, and writing more tokens than needed on the interface results in writing modulo the number of tokens needed (circular buffer). In Figure 2, the vertex B writes 3 tokens in a $Sink$ that consumes only one token, due to the circular buffer behavior, only the last token written will be made available to the outside world. This behavior allows to easily design iterative pattern without increasing the number of edges. This behavior can also lead to mistakes (from the designer view) as if there is no precedence between multiple occurrences of a vertex that writes to an output port, a parallel execution of these occurrences leads to a concurrent access to the interface and as a consequence to indeterminate data in the sink node. This also leads to dead codes from the node occurrences that do not write to the sink node.

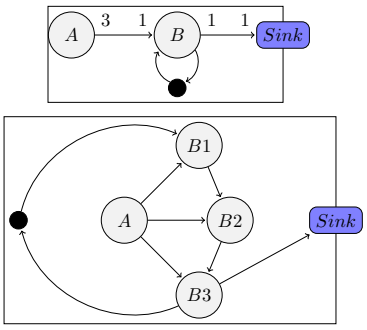


Figure 2. Sink example and its precedence graph

4.5 Interface-based hierarchy optimization for multi-core scheduling

Designing an application using a hierarchical model not only ease the designer work but also provide useful information about relevant way to group operation and data (operation and data clustering). In order to use those information for the mapping/scheduling step, only the top level of the graph is scheduled. This means that all the potential parallelism embedded in the hierarchy remains unavailable. In order to extract parallelism from the designer description, the graph must run through several transformation like HSDF transformations, and/or hierarchy flattening before being scheduled. To allow the designer to choose the level of potential parallelism to extract from the hierarchy, the graph hierarchy can be flattened to a given level thus revealing the potential parallelism in the parent graph. This means that a hierarchy level can be removed and the vertices it contains appear in the top graph. Running a HSDF transformation will then extract some more parallelism but will decrease the data-throughput since data sets will be divided in several smaller sets. Increasing the potential parallelism by flattening a hierarchy level also increases the number of vertices to schedule thus increases the scheduling complexity.

After those transformations the Interface-based hierarchy can be scheduled/mapped onto multi-core architecture using an adapted method.

4.6 Interface based hierarchy mapping and scheduling

By using off-line mapping and scheduling, one can generate, at compile-time, a deployment on a parallel architecture with optimized properties such as response time (latency), memory consumption or execution time. The inputs of the mapping and scheduling transformation are usually a Directed Acyclic Graph (DAG) of actors and a graph with processor vertices modeling the architecture [13]. The role of the mapping process is to choose one core to execute each actor while the scheduling process consists in choosing an order of execution for the actors.

The compile-time predictability of interface based SDF hierarchy enables the graph mapping and scheduling. A complete knowledge of all actors is necessary including Deterministic-Actor-Execution-Time (DAET), i.e. the time needed to execute each actor on each processor available for it. Before the mapping and scheduling process, a partial transformation of the graph in a Directed Acyclic Graph (DAG) must be applied, as described in section 2.1. Only the transformed highest level of hierarchy will be mapped and scheduled, reducing the mapping and scheduling complexity. The DAET of a hierarchical actor is computed from the execution time of the mono-processor scheduling of the actor's subgraph on the chosen processor. A trade-off between mapping accuracy and mapping speed is done while converting the interface based SDF hierarchy to a DAG.

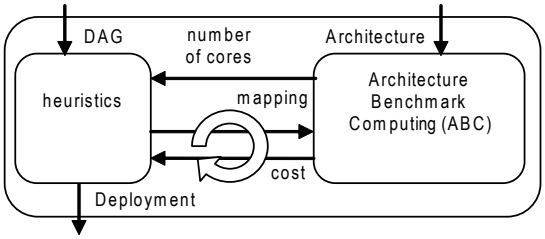


Figure 3. Structure of a split mapping scheduling algorithm

The problem of mapping and scheduling with the overall goal of minimizing the global response time, often simply called the task scheduling problem [13], has been proven to be NP-complete in [2] for realistic cases (more than 2 processors, actors with different timings, and so on). Heuristics are needed in order to solve the problem in polynomial time. In [7], the architecture of a mapping and scheduling processor is detailed where several heuristic are implemented and can be combined with several ways to evaluate a deployment. The mapping and scheduling algorithm is split into two parts, the heuristic part and the Architecture Benchmark Computing (ABC) part (see Figure 3). The heuristic part makes the mapping choices using an algorithm such as a list scheduling or a genetic algorithm. It minimizes a given cost and delegates the cost evaluation to the ABC part. The ABC part can return costs of any type: response time, memory consumption, execution time, etc... In the classic case of response time minimization, several ABCs are available to model the behavior of a deployment at different level of precision, taking into account the Inter-Processor Communication (IPC) impact.

After the mapping and scheduling process, a deployment is generated consisting in a DAG with mapping properties, a total order of the actors and added special Send and Receive actors that will be transformed into IPC calls during the code generation.

4.7 Code generation from interface based hierarchy

Code generation from an interface based hierarchy targets embedded application by generating static C code from a scheduling/mapping of the graph for a given architecture. One C file is generated for each core and the syntax must take advantage of available C code pattern in order to pro-

vide a human-readable code. To allow any C compiler to optimize the generated code, information such as variable scope and deterministic loop iteration domain must be taken into account. The generated C code must be architecture independent meaning that platform dependent code such as IPC functions, and operating system dependent functions will be generated as generic call that must then be implemented by the user. Additional properties on the graph, provide informations such as data types (integer, char ...), and each actor is associated to a C like prototype that gives information on the arguments types and order. Figure 4 gives an example of hierarchical graph and its mono-processor code generation.

4.7.1 Code generation from the top graph

After the mapping/scheduling step, the top graph is an SDF graph in which vertices have additional properties providing information on the core the vertex is mapped to and the vertex's scheduling order. This graph also have additional vertices representing Inter Processor Communications (IPC) as a pair of a send vertex and a receive vertex linked by a precedence arc.

Generating code from this graph consist in synthesizing the arcs into buffers and the atomic vertices into function calls. In our code generation arcs are synthesized as one-dimension arrays of a given type. Those buffers are then protected by semaphores when involved in IPC. Special vertices such as join, fork, broadcast, circular-buffer are synthesized as memory copy with relevant arguments : $Fork = \{N\} \Rightarrow N \times \{1\}$, $Join = N \times \{1\} \Rightarrow \{N\}$,$Broadcast = \{N\} \Rightarrow b \times \{N\}$, $Circular - Buffer = N \times \{1\} \Rightarrow \{1\}$. Vertices with repetition factor greater than one are embedded into a loop, which consume/produce from/in the input/output buffers by using pointers into those buffers (Figure 4). Atomic vertices are synthesized as function calls with their respective input/output buffers as arguments.

4.7.2 Code generation from a hierarchical vertex

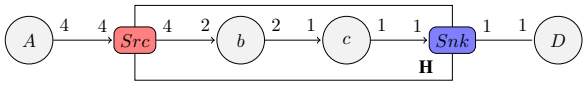
Generating code from a vertex which is associated to a hierarchy level is a little harder because we must ensure the interfaces to keep the same behavior as in the data-flow semantic. A pre-pass analyzes the interfaces behavior in the graph and adds broadcast vertex on source interface outgoing edges and circular-buffer vertex on sink interface incoming edge when needed.

- Broadcast adding rule : $c(e)\%p(e) \neq 0$
- Circular buffer adding rule : $c(e)\%p(e) \neq 0$

Those rules ensures that there will not be any buffer overflow when writing into a sink interface, and to not read out of the bounds of a source interface.

To simplify the code generation, strongly connected components in the hierarchy are clustered and edges containing delays are linked around the constructed cluster. This allows to synthesized strongly connected components in a factorized manner instead of repeating elements belonging to the strongly connected component. The scheduling order of the vertices in the block of code is obtained by doing a topological sort of the vertices.

As described earlier a hierarchy level behaves as a block of code meaning that all the arcs contained in it have a local scope. As a consequence a hierarchy level is described as a block of code in which arcs linking vertices are synthesized as local buffers, and arcs linked to interfaces are synthesized as pointer in a buffer obtained from the upper hierarchy level.




```
void main(void){
 int AtoH [4];
 int HtoD [1];
 int i ;
 while(1){
  A(AtoH);
  {
   int * src = &AtoH[0];
   int * snk = &HtoD[0];
   int btoc [4] ;
   for(i = 0 ; i < 2 ; i ++){
    int * sub_src = src[i*2%4];
    int * sub_btoc = btoc[i*2%4];
    b(sub_src, sub_btoc);
   }
   for(i = 0 ; i < 4 ; i ++){
    int * sub_btoc = btoc[i*1%4];
    int * sub_snk= snk[i*1%1];
    c(sub_btoc, sub_snk);
   }
  }
  D(HtoD);
 }
}
```

Figure 4. Example of a hierarchical graph and its code generation

5 Application case study

In this section we will show how the new hierarchy type (interface based hierarchy) can be used to design a IDCT2D_CLIP examples. The IDCT2D is a common application in image decoding which operates over a 8×8 matrix of coefficient to decode 8×8 pixel block. In the video decoding context the IDCT2D is followed by a clip operation which adds or not a sign bit on samples depending

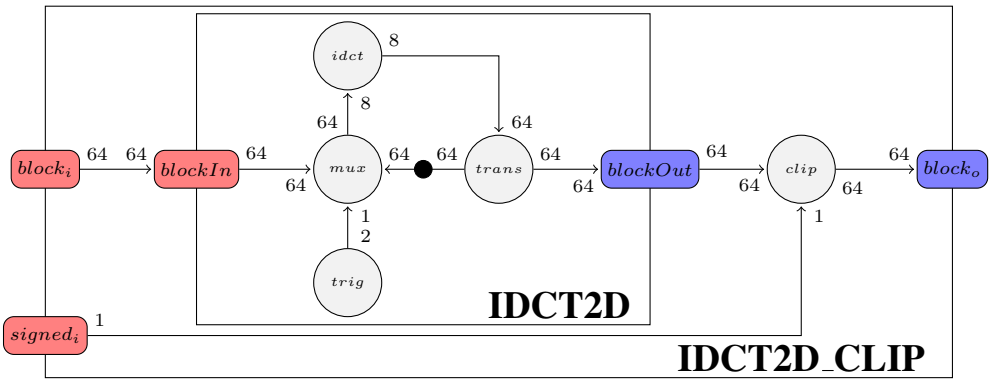


Figure 5. IDCT2D_CLIP SDF graph designed with hierarchy type 2

on the kind of prediction being used for the block (INTER or INTRA).

5.1 IDCT2D description

The IDCT2D_CLIP used in this example (Figure 5) is a recursive pattern using only 4 operations.

- *mux* : This actor, acts as a multiplexor. It outputs the data from the source port *blockIn* on its first firing and outputs the data from trans on the second firing.
- *idct* : Performs an one dimension IDCT on a vector of 8 elements (IDCT1D).
- *trans* : Transposes an 8×8 matrix.
- *clip* : Apply the signed depending on the kind of the prediction type.

In this representation the *trig* operation is a null time operation which forces the loop to iterate twice. The IDCT2D_CLIP is defined using two level of hierarchy. The first level performs a classic IDCT2D by using one dimension IDCT and transposition of an 8×8 matrix. The additional level add the *clip* operation which is specific to the video decoding algorithm. This operation computes on each sample a 9 bit signed integer for INTER prediction, while it does an 8 bit unsigned integer for INTRA prediction.

5.2 Example Mapping/Scheduling

In order to show the result of the IDCT2D mapping on a four cores architecture, the application is instantiated into an application generating four blocks of data to be computed and a single signed data. The IDCT2D vertex is thus triggered four times and the signed data is broadcasted to each instance of the IDCT2D. In order to extract some parallelism from the description, an HSDF transformation is applied, which instantiates four times the IDCT2D vertex, an adds fork and join vertices on the input and output data. The mapping step distributes an IDCT2D on each core of the architecture and generates the send and receive operation on the useful data. The result shows that the mapping takes advantage of the hierarchy by generating grouped data transfer thus minimizing the communication setup overhead in the IPCs. Flattening the hierarchy leads to a larger number of vertex to schedule but the mapping leads to the same solution, and the code generation gives a bigger application memory footprint since all the buffer embedded into the hierarchy are allocated as global buffers. Figure 6 gives statistics extracted from the scheduling. The DAG size refers to the number of vertices in the DAG to schedule, and is given for different hierarchy flattening level. The memory footprint refers to the buffers allocated statically. The buffers embedded into hierarchy levels, are allocated dynamically by the compiler and takes advantage of their respective scope (code block). The memory statistics presented are deduced from the code generation.

	HSDF	Flattening Level 1	Flattening Level 2
DAG size	9	13	33
Memory	323 int	385 int	1092 int

Figure 6. Applications statistics for different level of flattening : static memory allocations per core and number of vertices in the DAG

6 Further work and Conclusion

This paper introduces a new form of hierarchy semantics for synchronous dataflow (SDF) representations that involves the designer more in the application optimization process by allowing him to modify the topological descriptions of application subsystems with more freedom relative to the corresponding subsystem interfaces. The ap-

```
void idct2d_clip(int * blocki,
                 int *sign,
                 int * blocko){
  int block_out[64];
  int triggers[2];
  int idct1d_loop[64];
  init_delay(idct1d_loop,64/*init_size*/);
  trig(triggers);
  for(i = 0; i<2 ; i ++){
    int *trigger = &triggers[(i*1)%2];
    int rows_in[64];
    int rows_out[64];
    mux(blocki,idct1d_loop,trigger,rows_in);
    for(j = 0; j<8 ; j ++){
      int *row_out = &rows_out[(j*8)%64];
      int *row_in = &rows_in[(j*8)%64];
      idct1d(row_in,row_out);
    }
    trans(rows_out,block_out,idct1d_loop);
  }
  clip(block_out,sign,blocko);
}
```

Figure 7. Code generated using PREESM

plication of our methods to multi-core DSP system design is demonstrated with a detailed case study involving two-dimensional discrete cosine transform computation.

The new form of SDF hierarchy that we propose in this paper allows the designer to perform optimization on the application at a topological level (i.e., in terms of rearrangements of dataflow block instances and their interconnections), and provides a programming interface for hierarchical organization that is more natural in various contexts. In particular, our hierarchy representation is closer to the semantics of C and other common target languages for SDF-based synthesis flows, and makes the application easier to describe for programmers who are, for example, more familiar with C, and less familiar with concepts such as repetitions vectors and subinit graphs. Our method allows reuse of graphs developed in other applications with minimal modifications, and offers more flexibility by allowing the description of execution patterns that do not map directly into conventional types of hierarchy. Our proposed interface-based hierarchy semantics for SDF graphs has been implemented as the algorithm specification model in the PREESM tool [9].

A useful direction for further investigation is the development of techniques for optimized scheduling that are derived from our proposed new form of SDF hierarchy organization. Such scheduling could automatically optimize execution for the given SDF graph by choosing to remove hierarchy levels or perform SDF-to-homogeneous-SDF transformation on selected subsystems to take advantage of available parallelism. Code generation could also be improved by integrating target compiler directives with code at subsystem boundaries that back-end compilers can use for more thorough optimization.

References

[1] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[3] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, June 2003.

[4] C. Hsu, J. L. Pino, and S. S. Bhattacharyya. Multithreaded simulation for synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*, pages 331–336, June 2008.

[5] E. Lee and D. Messerschmitt. Pipeline interleaved programmable dsp's: Synchronous data flow programming. *Proceedings of the IEEE*, 35(9):1334–1345, Sept. 1987.

[6] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[7] M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan. Scalable compile-time scheduler for multi-core architectures. In *Design, Automation and Test in Europe Conference (DATE)*, 2009.

[8] J. Piat, S. S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for Synchronous Data-Flow Graphs. *in Signal Processing Systems (SiPS)*, 2009.

[9] J. Piat, M. Raulet, M. Pelcat, P. Mu, and O. Déforges. An extensible framework for fast prototyping of multiprocessor dataflow applications. In *IDT08: Proceedings of the 3rd International Design and Test Workshop*, december 2008.

[10] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling system for DSP applications. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 122–126 vol.1, November 1995.

[11] H. W. Printz. *Automatic mapping of large signal processing systems to a parallel machine*. PhD thesis, Carnegie Mellon University, 1991.

[12] G. C. Sih and E. A. Lee. Dynamic-level scheduling for heterogeneous processor networks. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 42–49, October 1990.

[13] O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, 2005.

[14] W. Sung, M. Oh, C. Im, and S. Ha. Demonstration Of Codesign Workflow In PeaCE. In *in Proc. of International Conference of VLSI Circuit*, 1997.